The Digital Anatomist Foundational Model Server

Darren Stalder and James F. Brinkley
Structural Informatics Group
Dept. of Biological Structure, Box 357420
University of Washington, Seattle, WA 98195
darren@u.washington.edu
brinkley@u.washington.edu
http://sig.biostr.washington.edu/

July 1, 1999

1 Introduction

This paper is intended to show the design decisions, tradeoffs and advantages of implementing a complex server using Perl.

Many people don't recognize Perl as being a full-blown application language; they seem to think that Perl is only useful for small jobs or web scripts. Perl can do much, much more than that. In this case, it works very well for a complex stand-alone server.

A Perl implementation requires different choices than C. Perl has the benefits of loose typing, dynamic code execution, and high-level opcodes, but there are tradeoffs. C offers more direct control, less memory consumption, and faster code. For most servers though, the benefits of using Perl far outweigh the costs. Sheer speed of writing code means that you will have more time to tune the server to your specific needs.

This paper discusses the implementation of a server written for the Digital Anatomist Project at the University of Washington. The goal of the Project is to build a knowledge-based anatomy information system (AIS) which permits the re-use of anatomical information in multiple Internet-based applications in clinical medicine, education, and research [BWHR99].

The primary source of knowledge in the AIS is the Foundational Model of Anatomy (FM), a symbolic abstraction that explicitly declares the principles and concepts necessary to coherently and consistently model anatomical knowledge [RSB98]. The FM symbolically describes all structures visible to 1 mm resolution, together with their semantic relationships such as ontological classification, parts, and branches. The horizontal plane of Figure 1 show the first two FM levels in the anatomical ontology (Physical Anatomical Entity → Anatomical Structure, Body Substance, Anatomical Spatial Entity); (Anatomical Spatial Entity → Anatomical Junction, Anatomical Landmark....). The vertical dimension shows the mapping of some of these anatomical classes to a parallel spatial ontology.

The FM is currently represented as a semantic network, a type of graph in which the nodes represent the classes in the various ontologies (e.g. Anatomical Junction, Body Region), and various kinds of links represent the semantic relationships among these classes (isa, part-of, branch-of, etc.).

The FM is stored as two tables in a relational database: a terms table representing the nodes, and a links table representing the links. During start-up the server reads these two tables and turns them into an in-memory multi-dimensional graph. Each node in the graph has attributes, such as Author and Authority, as well as parent-child relationships in the different hierarchies. The current hierarchies are isa, part-of, branch-of, tributary-of, and contained-in.

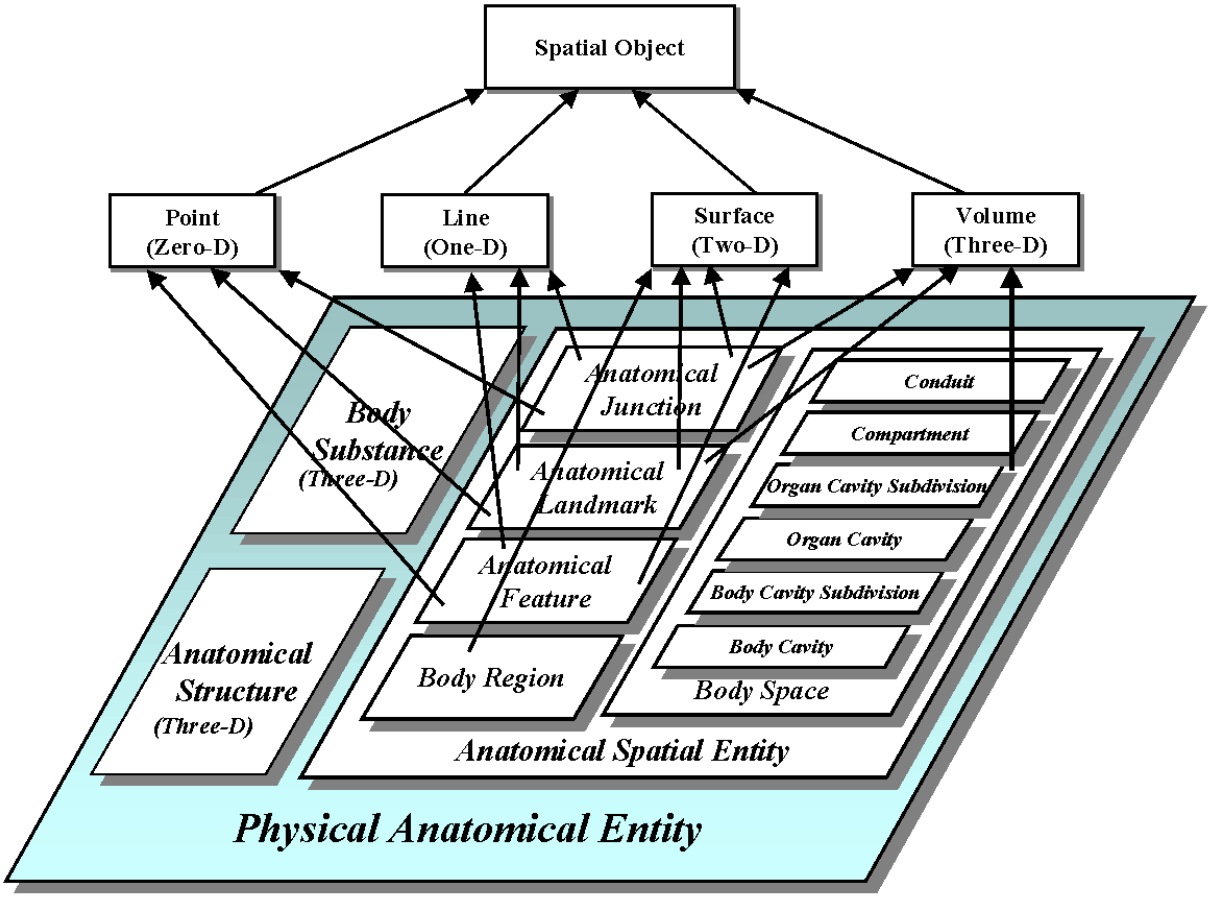

Figure 1: First three levels of Foundational Model of Anatomy graph

Each node may also have directed subnodes that are synonyms of the main node. Synonyms share the graph relations of their parent node but have their own attributes.

Each of these nodes is uniquely identified by a UWDAID (University of Washington Digital Anatomist IDentification). The names of the nodes and sub-nodes may change and swap. The names might swap when what appeared to be the preferred name for a node can't be made consistent with other parts of the Foundational Model. At that point, the more consistent name is made the preferred name and the old name becomes a synonym. Sub-nodes may be created and destroyed but the UWDAID stays the same. A UWDAID may not be changed, only destroyed. In destroying the UWDAID, that node and all its sub-nodes are destroyed as well. That ID will never be reused.

2 Server Implementation

Background

The original Foundational Model was implemented on the NeXT computer, with a GUI frontend on top of Sybase [BR97]. While NeXTStep and Sybase made this easy to implement, access to the FM was restricted; only NeXTStep clients could talk to it. No other tools could make use of the knowledge being entered into the system. The information needed to be opened up for other client applications. This was accomplished by writing the first C-based version of the Foundational Model Server (FMS), implementing a new middle-layer that accessed the same backend Sybase database as the NeXTStep GUI. We chose a lisp-like protocol for the FMS because many of our other applications are lisp-based [BP97].

This new middle layer was very useful since any application that talked the protocol could access the FM. For example, a scene builder [WB98] used the FM to group 3-D anatomical models into 3-D scenes. The middle layer also provided a better interface to the data. All changes to the data were checked for consistency and validity. Also, this allowed the data to be represented from a knowledge/ontological perspective rather than simply as queries on a database. We went from applications calling:

```
select t.Name, l.seq from terms t, links
  l, terms u where t.UWDAID = l.ChildID and
  t.Role = 'Preferred Name' and u.UWDAID =
  l.ParentID and u.Name = 'Heart' order by
  l.seq
```

to

```
(fm-get-children "Heart" "part of").
```

However, this server didn't scale up well. After about 15,000 terms were reached the application became slower and slower until simple queries were measured in terms of minutes. To deal with the speed issues, it was necessary to rewrite the FMS. It went from being written in C on a NeXT system to being written in Perl on a Linux box. The original server was written as a series of SQL calls underneath the middle layer, which led to slowness and inflexibility in writing new knowledge query functions. The new Perl-based server pulls all the nodes into memory, and works with them the same way that it presents them to the clients. Now, not only is access to the knowledge much faster, but new knowledge query functions are simplified since they don't have to be translated into SQL first.

Current Perl Server

The server code is split into three distinct perl modules: `fms.pl` (the functional interface), `Server.pm` (the forking server), and `DBI_if.pm` (the interface to DBI). With this division both the server model and database model can be changed without modifying `fms.pl`, the client-visible core of the server.

fms.pl

`fms.pl` is the entry point for the FMS. It is responsible for all control and the functional implementation of the server. Using Perl's self-reflection, new functions can be added to the FMS with no dispatch tables and no changes to the support functions. A new function can just be written in `fms.pl` as normal code. When an API call is made, `fms.pl` checks the validity of the call. If it checks out, the function requested is then invoked with its parameters. There's no need to keep any arrays of function pointers. The

help function likewise can just look into the %fms:: stash and list those functions available for use by the programmer. Internal functions are hidden from the help system when their names start with an underscore.

To start the system, fms.pl first runs Server::setup to allocate the socket and detach itself from the terminal. Once that is finished, it calls DBI_if::setup to initialize all data structures. The main hashes and some helper variables are exported back into fms.pl. Finally, fms.pl calls Server::MainLoop. This runs until the server exits.

Server::MainLoop sits on the socket waiting for clients. When a client comes in, Server::MainLoop validates the client and calls fms::service_request. service_request then handles the rest of the communication with the client. Occasionally when someone makes a change to the knowledge-base, fms.pl calls the appropriate DBI_if::* function to write the data to the database.

The Foundational Model Server has been broken up in this way so that the implementation details are easy to change. fms.pl shouldn't ever have to change because of implementation or platform; those details are all handled by Server.pm and DBI_if.pm. The requirements that fms.pl has of these two is light.

DBI_if.pm

DBI_if.pm has a setup function to populate the data structure hashes. It also provides a small number of callbacks so the data that the client changes in memory can be pushed on to the database. If the only implementation that needs to be changed is to a different database, DBI makes that easy. In this case all that is needed is changing the use DBD::*; line at the top of the file and to change the DSN in the DBI->connect call.

Server.pm

Server.pm needs to supply a little bit more than DBI_if.pm but not much. It manages a persistent process that serves multiple clients coming in over a socket. Server.pm could do this as a forking daemon process, a select(2)-based server, or a multi-threaded application. It wouldn't matter to fms.pl.

Server.pm must also provide two different atomically increasing numbers. The first is a unique UWDAID. This needs to be unique over many different clients and it is vital that no number is ever re-used.

The other unique number is used for client synchronization. Every authoring operation first generates the Perl code that will change the internal hash structures to the new state. This code is then passed to a function that evaluates the code for the current client and writes out a file for all other clients to read so that their state may be updated as well. The name of this file is the second unique number. We don't want two clients creating the same filename and writing over each other, therefore this filename must be unique among all the clients. At the beginning of the accept(2) loop for the server, the value of this number is checked. If it is greater than the currently stored value, the server knows that some authoring operations have occurred. It reads and evaluates the code in each file that it finds until its internal value matches that number. It is now ready to fork off new clients with the new structures. The PID of the client is stored in each update file. This is so that later, client updating may occur. Each client would read the files and skip those where the PID match their own. An example file that the update code writes out is in Figure 2.

Propagation of the data from the children of the server out to the client applications is considered to be a future project. "Wetware" co-ordination is currently used. When the client-data needs to be updated, the user can disconnect and reconnect, thereby retrieving fresh data. This method of data synchronization by running dynamically generated code couldn't be done in most languages. If this was C or C++, a mini-language would have to written describing updates that an interpreter would then have to read and evaluate. The dynamic nature of Perl allows the server to generate and run first class code in its process space. The time freed up with this is liberating.

By using semaphore sets with IPC::SysV, it is trivial for Server.pm to create and manage the unique numbers mentioned above using semaphores. All children know the keys of the two semaphore

```
392
Wed Jun 23 09:35:35 1999

# delete the Preferred Name from the Name hash
delete $Name{lc "Body of fifth sacral segment"};
# delete all the Synonyms from the Name hash
delete @Name{map { lc $_ } keys %{$Master{31857}}}
    if exists($Master{31857});

# delete the parents and the children
my @hier = keys %{$ID{31857}{Hierarchy}};
foreach my $hierID (@hier)
{
    foreach my $parentID (get_links(31857, $hierID, "Parents"))
    {
        delete $ID{$parentID}{Children}{"31857 $hierID"};
    }
    foreach my $childID (get_links(31857, $hierID, "Children"))
    {
        delete $ID{$childID}{Parents}{"31857 $hierID"};
    }
}
# delete the ID from each hierarchy it's in
foreach my $hierID (@hier)
{
    delete $Hier{$hierID}{31857};
}

# finally delete the term from the ID hash
delete $ID{31857};
```

Figure 2: Dynamically generated code to delete the concept, fifth sacral segment

sets, having received them from their parents. From there, they can lock the semaphore, wait on it if necessary, increase the value by one, store that value, and unlock the semaphore. Care is taken in the specification of the semaphore operations so that if the process locking the semaphore dies, the operating system will automatically unlock the semaphore and undo the current operation during normal child cleanup. Since this is a common operation, we encapsulated it in a function in `IPC::Semaphore` and sent it off to Graham Barr for possible inclusion in the master version of `IPC::Semaphore`. With the Open Source nature of Perl and its modules this is easy and gratifying.

In designing `Server.pm`, we chose to go with a forking multi-client model over using select(2) or multi-threading. Using fork(2) allows each process to have an autonomous process space that does not disturb the master server or the other clients. There are no worries about memory-pollution, hanging processes, or dying clients. It just works. Considering that most modern unices utilize copy-on-write, forking doesn't cause much of a problem. Here is the one place we found that using Perl causes something of a problem. We have large, deep hash structures, most of which don't change over the life of the client connection. Unfortunately, Perl has no way of designating these as read-only or static. So, when more memory is allocated for more structures, some of the static hash structures are re-arranged and the memory savings are lost.

3 Use of the Server

The high level API that the FMS implements has allowed clients to be written in Perl, C, Lisp, and Java. These clients are in use in authoring and end-user applications in clinical medicine and education, both locally and at other universities. The original clients of the NeXT-based server now talk to the FMS with no modification to them.

The Foundational Model Builder (see Figure 3), written as a Java applet, is our primary client application, and is in use by the Structural Informatics Group to build a complete, consistent, and detailed symbolic structure of human anatomy. Various collaborators from Stanford, Columbia, Freiburg University — Germany, Harvard, and others use the same applet to view and make comments on the Foundational Model. Since we're using tcp sockets and a forking multi-client server, this is easy. We just point them to the webpage and the applet connects to the server from their remote site. Other Java applications are being written to view the anatomical data in different ways.

Another client we have written is a Perl module for accessing the FMS. An example client is shown in Figure 4. The output of this client is:

```
$ fm-print-kids "Long bone" "part of"
Long bone
  Compact bone of long bone
  Trabecular bone of long bone
  Cartilage of long bone
  Periosteum of long bone
  Medullary cavity of long bone
  Diaphysis
    Diaphysis proper
      Compact bone of diaphysis proper
      Trabecular bone of diaphysis proper
      Periosteum of diaphysis proper
    Metaphysis
      Compact bone of metaphysis
      Trabecular bone of metaphysis
      Periosteum of metaphysis
    Compact bone of diaphysis
    Trabecular bone of diaphysis
    Periosteum of diaphysis
  Epiphyseal plate
  Epiphysis
    Compact bone of epiphysis
    Trabecular bone of epiphysis
    Periosteum of epiphysis
```

Compare this to the screenshot of Figure 3. Clients using this module are being developed with inferential knowledge for automated entry of recurrent terms such as the highly similar structures of the vertebrae.

Other applications using the FMS include a radiologic treatment planning system [KWL+99] written in Common Lisp for PRISM, an anatomical image-annotation system [LB99] written in Java, and a 3-D anatomical scene generator [WRB99] written

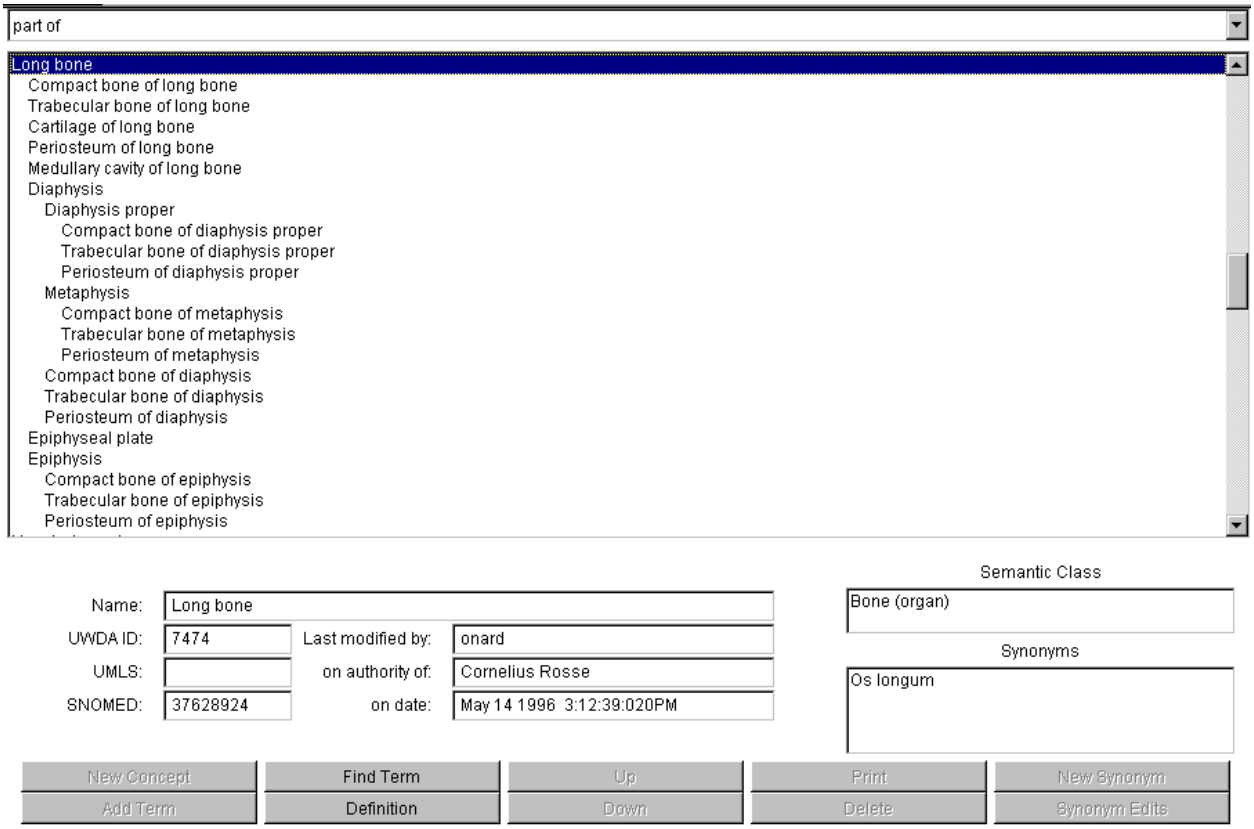

Figure 3: Foundational Model Builder

```
#!/usr/bin/perl -w

use Net::Telnet 3.01;
use FMS;
use strict;

my ($t, @lines, $lines);

@ARGV == 2 || die "Usage: $0 <term> <hier>\n";

my($term, $hier) = @ARGV;

$t = new Net::Telnet(Binmode => 0, Telnetmode => 0, Timeout => 75,
                     Output_record_separator => undef);
$t->open(Host => 'fms', Port => 'fms');
$t->waitfor('/.*\003/');

contains($term, $hier) || die "'$term' doesn't exist in '$hier'\n";

print_all($term, $hier);

sub print_all
{
    my ($term, $hier, $level) = @_;

    print "$term\n" if !defined($level);
    $level ||= 1;

    foreach my $child (get_children($term, $hier))
    {
        print "  " x $level, $child, "\n";
        print_all($child, $hier, $level + 1);
    }
}
```

Figure 4: Example Perl client using the FMS.

using Perl-CGI and our lisp-based graphics language [BP97]. Future projects include interfacing the FMS with a Protégé front-end [HBB+99], a Medical Illustration Toolkit, and a Java 3-D environment called "The Virtual Playground" [SBC+98].

4 Where to now?

There is widespread interest in the Foundational Model of Anatomy. More and more applications are being written to the FMS API. Some extensions include making the FMS talk using an XML protocol and/or using the Protégé interface mentioned above. Areas of development for the server include inference-based authoring, select(2)-based access for speed, node-level locking, and a reduction in memory requirements.

The FMS expects the authors of the FM to make no errors and be consistent throughout their input. Since this is unrealistic, errors and consistency checks are run during a post-processing phase. The authors would prefer the FMS to validate their input in real-time. It should check spelling as well as consistency of the name syntax e.g. "Fifth vertebral arch" vs. "Arch of the fifth vertebra".

Currently each term must be entered into the FMS even though it may be a repetition of existing terms. The FMS should supply matching terms as needed. For example, the third through seventh cervical vertebrae are highly similar. Using inference rules and a template cervical vertebra, the FMS would return the dynamically built cervical vertebral structures and their children.

When a client connects, the master server forks off a child that handles all transactions for that connection. In general, that is good since most clients will stay connected for a period of time. In the case of connect-and-exit clients such as CGI scripts, however, this doesn't work well. One child could be spawned off that handles all connects through a select(2) loop. This would be much faster but could hang or die if something goes wrong. The master server would notice this and start another child to replace the old one.

All the authoring on the Foundational Model takes place at the University of Washington. While the authors sometimes work from home, they still co-ordinate with each other daily. This is important since there's a small chance that the data structures in the master server could be corrupted if two authors were working on the same term at exactly the same time. It would be useful if the server did the locking and co-ordination itself. This shouldn't be too difficult since it's a solved problem for databases already.

The server uses 45 Megabytes of memory at this time. On modern server systems, this isn't much of a problem. It does restrict the server from being demonstrated on most laptops though. These memory requirements can only be expected to rise as more terms are entered. Such memory requirements are from *all* the data being read into memory. Many of the data-structures don't *have* to be in memory; it's just convenient. If the nodes and their relationships are kept in memory and the attributes retrieved as needed from the database or an LRU cache, the memory limitations could be addressed without sacrificing speed.

5 Conclusion

We have described a Perl-based standalone server that acts as a middle layer between a backend database and multiple client applications. The server, called the Foundational Model Server (FMS) provides an abstract protocol for communicating with a large foundational model of anatomy that hides the details of the particular database implementation, and allows error checking, multi-author synchronization and caching for fast response time. The FMS is becoming widely used for multiple and distributed applications of an anatomy information system. The use of Perl as an implementation language has greatly decreased the development time and has demonstrated that Perl can be used for much larger applications than small jobs or simple CGI scripts. Since the FMS is independent of content it should be useful for the implementation of other Foundational Models, and eventually for linking Foundational Models developed at diverse sites. The current implementation of the FMS can be downloaded from `http://sig.biostr.washington.edu/software/downloads/fms.tar.gz`.

Acknowledgements

This work was funded by National Library of Medicine grant LM06316.

References

[BP97] J. F. Brinkley and J. S. Prothero. Slisp: A flexible software toolkit for hybrid, embedded and distributed applications. *Software – Practice and Experience*, Vol. 27:pp. 33–48, 1997.

[BR97] J.F. Brinkley and C. Rosse. The digital anatomist distributed framework and its applications to knowledge based medical imaging. *Journal of the American Medical Informatics Association*, 4(3):165–183, 1997. Also published in van Bemmel, J.H. and McCray, A.T. (eds.) Yearbook of Medical Informatics 98, pp 119-137, 1998.

[BWHR99] J.F. Brinkley, B.A. Wong, K.P. Hinshaw, and C. Rosse. Design of an anatomy information system. *Computer Graphics and Applications*, 19(3):38–48, 1999.

[HBB+99] J. S. Hahn, E. Burnside, J. F. Brinkley, C. Rosse, and M. A. Musen. Representing the digital anatomist foundational model as a protege ontology. In *Proceedings, American Medical Informatics Association Fall Symposium*, page In Press, Washington, D.C., 1999.

[KWL+99] I. J. Kalet, J. Wu, M. Lease, M. M. Austin-Seymour, J. F. Brinkley, and C. Rosse. Anatomical information in radiation treatment planning. In *Proceedings, American Medical Informatics Association Fall Symposium*, page In Press, Washington, D.C., 1999.

[LB99] W. B. Lober and J. F. Brinkley. A portable image annotation tool for web-based anatomy atlases. In *Proceedings, American Medical Informatics Association Fall Symposium*, Washington, D.C., 1999.

[RSB98] C. Rosse, L. G. Shapiro, and J. F. Brinkley. The digital anatomist foundational model: principles for defining and structuring its concept domain. In *Proceedings, American Medical Informatics Association Fall Symposium*, pages pp. 820–824, Orlando, Florida, 1998.

[SBC+98] P. Schwartz, L. Bricker, B. Campbell, T. Furness, K. Inkpen, L. Matheson, N. Nakamura, L.-S. Tanney, and S. Yeh. Virtual playground: Architectures for a shared virtual world. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, pages pp. 43–50, New York, 1998.

[WB98] B. A. Wong and J. F. Brinkley. Dynamic 3-d scene navigation in web-based anatomy atlases. In *Proceedings, American Medical Informatics Association Fall Symposium*, page pp. 1100, Orlando, Florida, 1998.

[WRB99] B. A. Wong, C. Rosse, and J. F. Brinkley. Semi-automatic scene generation using the digital anatomist foundational model. In *Proceedings, American Medical Informatics Association Fall Symposium*, page In Press, Washington, D.C., 1999.